LOW LATENCY STOCHASTIC FILTERING SOFTWARE FIREWALL ARCHITECTURE

A Thesis

by

PRITHA GHOSHAL

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Paul V. Gratz
Committee Members,	Alex Sprintson
	A L Narasimha Reddy
	Vivek Sarin
Department Head,	Costas N Georghiades

December 2012

Major Subject: Computer Engineering

ABSTRACT

Firewalls are an integral part of network security. They are pervasive throughout networks and can be found in mobile phones, workstations, servers, switches, routers, and standalone network devices. Their primary responsibility is to track and discard unauthorized network traffic, and may be implemented using costly special purpose hardware to flexible inexpensive software running on commodity hardware. The most basic action of a firewall is to match packets against a set of rules in an *Access Control List* (ACL) to determine whether they should be allowed or denied access to a network or resource.

By design, traditional firewalls must sequentially search through the ACL table, leading to increasing latencies as the number of entries in the table increase. This is particularly true for software firewalls implemented in commodity server hardware. Reducing latency in software firewalls may enable them to replace hardware firewalls in certain applications. In this thesis, we propose a software firewall architecture which removes the sequential ACL lookup from the critical path and thus decreases the latency per packet in the common case. To accomplish this we implement a Bloom filter-based, stochastic pre-classification stage, enabling the bifurcation of the predicted good and predicted bad packet code paths, greatly improving performance. Our proposed architecture improves firewall performance 67% to 92% under anonymized trace based workloads from CAIDA servers. While our approach has the possibility of incorrectly classifying a small subset of bad packets as good, we show that these holes are neither predictable nor permanent, leading to a vanishingly small probability of firewall penetration.

DEDICATION

To my family

ACKNOWLEDGEMENTS

I would like to thank my family for all the encouragement, inspiration and support they have given me all through these years without which it would not have been possible for me to reach this level.

I am grateful to my professors at my undergraduate college National Institute of Technology Karnataka, for initiating in me the desire to pursue future studies in Computer Architecture as well as endowing me with the basics of the knowledge required for the same.

I would not have been able to work on this project without the help and guidance of Dr. Paul V Gratz, Dr. Alex Sprintson. I could also not have done this work without the introductory class that I had taken under Dr. A L Narasimha Reddy. I also thank Dr. Vivek Sarin for his efforts in revewing and evaluating my research work.

This research would not have been possible without Jasson Casey. Jasson has been there all the time to help by giving hints for all doubts that I had, thus helping me in a way so that I could still learn the concepts by myself. I also thank Andrew David Targhetta for all the discussions that we had during the formulative phase of this project. Also, I would like to thank Rajesh Kumar, for staying by me, discussing with me and helping me cheer up even if things were not going so well.

Finally I would like to thank Texas A&M university and the Department of Electrical and Computer Engineering for giving me an opportunity to pursue my Masters degree.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Network security is an increasingly critical part of network design and implementation. Network security involves the planning of an organization's network infrastructure to protect applications, sensitive data and resources from unauthorized accesses. Network attacks are constantly increasing, with browser based attacks hitting nearly 1 billion incidents in 2011 [1]. These growing attacks make it imperative to have better and faster security mechanisms which can perform at line speed while preventing attackers' view into the protected system. Network firewalls are a basic component of any network security implementation which attempt to address the problem of malicious network traffic. In this thesis we present a novel approach to rearchitecting software firewalls on commodity hardware, with the aim of making software firewalls competitive with custom hardware firewalls.

There are many different ways in which a device in a network can be attacked by malicious devices, but they can be broken down into the following three main categories:

1. *Intrusion* - This occurs when someone tries to use a device posing as a legitimate user.

2. *Denial of Service* - An attack aimed to prevent the availability of resources.

3. *Information theft* - The exposure of confidential information or personal information to unauthorized persons.

Denial of service(DoS) attacks are one of the most common attacks and every year sees an increase in the number. During the fourth quarter of 2011, the number of

distributed DoS attacks increased by 45% compared to the similar period in 2010 and more than twice the number of attacks in the third quarter of 2011 [2].

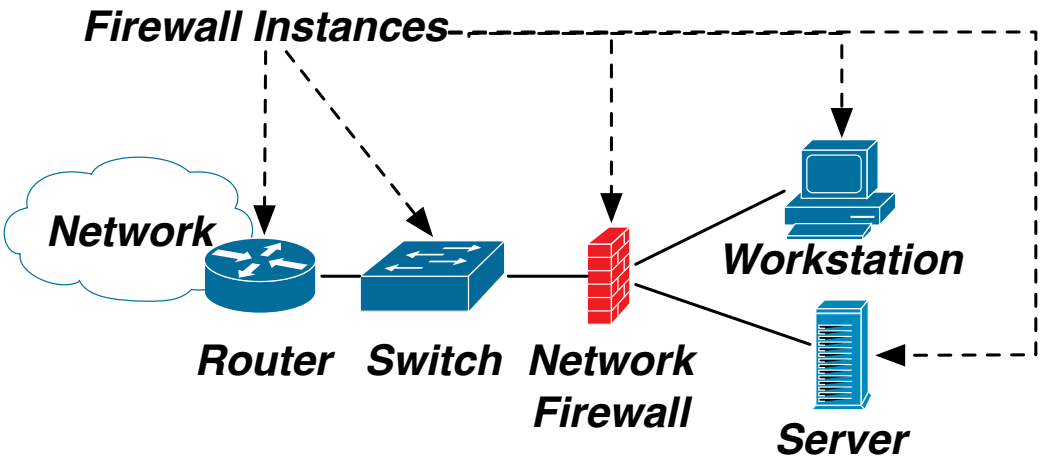


Fig. I.1. An example of firewall instances

As illustrated in Figure I.1, firewalls regulate and validate the flow of network traffic between hosts and clients in wide-area networks (WANs) or the Internet and a given private local area network (LAN). A firewall may be implemented as a custom hardware device, or as software running on commodity compute hardware. Firewalls form part of the gateway to the devices/computers inside the private LAN, ensuring the LAN's security from malicious external attacks. It examines all incoming packets from the Internet and depending on the various fields identifying the packet, it decides whether to allow the packet or discard it.

In order to ensure packets are not dropped arbitrarily, an effective firewall must be fast enough to process incoming packets at their arrival rate. At present, commodity hardware systems function significantly slower than the line speed of current, top of the line, 10Gbps ethernet cards. Where full line speed processing is required, fast, custom hardware solutions are often used. These hardware solutions, however, are

power hungry and extremely expensive.

We propose a firewall architecture which is implemented as a part of a standard Linux kernel and which significantly closes the performance gap between software and custom hardware-based firewalls. The architecture makes use of a Bloom filter, a highly efficient, low-complexity, probabilistic data structure, as a preclassifier to divide the stream of incoming packets into-

- Probably known good packets
- Bad packets or Previously unseen good packets

While the data structure is stochastic, i.e. yielding a small percentage falsely classified "good" packets, we will show that this percentage is both vanishingly small and unpredictable to an attacker in our setup. After the preclassifier, cache-like data structures are used to ensure that a slow ACL throughput will not limit firewall performance. Only the packets which are not found in the caches, are sent to the ACL, the result from which updates either the preclassifier or one of the caches.

The proposed firewall architecture has the following properties:

- *Remove the ACL from the critical path* - The main sequential ACL lookup is removed from the critical path. Instead a constant time search through a stochastic data structure (a Bloom filter) of order O(1) is used. The ACL search is placed on a non-critical path to update the data structure. This approach speeds the critical path (allowed packets path), preventing some types of Denial of Service attacks.
- *Decoupled the Allow and Deny paths* - The packets on the *allowed* path can proceed irrespective of the packets on the *deny* path.

The rest of the thesis is as organized as follows: Chapter II goes over the previous work in this area, Chapter III gives some background on ACLs, Bloom filters and motivation of our work, the proposed idea is explained in Chapter IV, followed by Chapter V containing the methodology and evaluation. Finally, Chapter VII concludes.

CHAPTER II

PREVIOUS WORK

Firewalls are a crucial part of network security. Since their advent, researchers have explored on the various aspects of firewalls, which we divide into three main sections:

1. *Firewall Policy* - the design of firewall rules in the ACL and making them more robust, accurate and easy to design.

2. *Search Algorithm* - much research focuses on speeding the ACL search using various data structures and algorithms.

3. *Custom hardware firewalls* - custom hardware based firewalls have included *caches* and hardware based solutions like $TCAMs$.

Each of these topics are explored in details below.

II-A. Firewall Policy

Firewall policy involves the design on the firewall rules. Gouda and Lui examine the structure of firewall rules with an aim towards improving accuracy [3]. Others have studied how conflicting rules can be detected and presented to the firewall designer for removal or rearrangement [4, 5, 6]. To analyze the functionality of the firewall, the firewall policies can be queried like a database [7]. High level languages to specify firewall rules which increase the level of abstraction have also been proposed [8, 9, 10, 11].

These approaches are orthogonal, and potentially complementary to our work. Even with optimal ACL rules, search algorithm complexity will limit performance of software firewalls on commodity hardware.

II-B. Search Algorithm

An important body of research examines specialized data structures to improve firewall performance. This encompasses not only firewalls, but also route classification (e.g. routing of outgoing packets) and other network applications. The generic problem is known as packet classification. Packet classification can be defined as a mechanism to categorize packets based on one or more fields in the packet header. Increasing network bandwidth has given rise to a requirement of fast packet classification which can keep up processing with line speed in the order of Gbps.

In typical software firewalls, classification throughput is directly related to the complexity of the ACL. A naive, linear algorithm will perform very poorly on large, complex ACLs due to sequential matching required.

Gupta and McKeown define a taxonomy of packet classification algorithms which has the following categories [12].

- Basic Data Structures:
  - *Linear search* - The simplest implementation is a linked list of rules in order of decreasing priority. The rules are checked one by one till there is a match. This is simple and storage efficient but the time taken grows linearly with the number of rules. This is the baseline search algorithm used in the standard Linux kernel.
  - *Hierarchical tries* - A D dimensional hierarchical trie is composed as follows, the first trie is built based on one of the fields in the packet header, the nodes of this trie gives rise to individual tries based on different fields in the packet header recursively. For example, one trie can be formed based on the source address, and each node may give rise to a trie based on the

destination address. Each rule is stored exactly once [12].

  - *Set Pruning tries* - This is similar to a hierarchical trie but rules are replicated to reduce the latency of traversal [13].

- Geometry-based Structures:

  - *Grid of tries* - Srinivasan et al. proposed a grid of tries data structure for 2D packet classification [14]. This is similar to the Hierarchial trie in that each rule is stored only once. To reduce the latency of search in some of the trie nodes, a switch pointer is stored to guide the search process.

  - *Cross-Producting* - Cross-producting is done by classifying packets based on separate 1 dimensional lookups and then combining the results to get the final decision. This can be used for an arbitrary number of dimensions [14].

- Heuristic Approaches:

  - *Tuple space search* - The search space is broken up into tuples based on the number of specified bits in the rules (leaving the don't care bits [15]). The search then takes place inside a tuple using hashing.

  - *Hierarchial Intelligent Cutting (HiCuts)* - The algorithm divides up the search space in each dimension (that is for each field in the header over which search is performed) [16]. The final search is performed over a small number of rules.

II-C. Custom Hardware-based Firewalls

Software based firewall architectures include caching to prevent all requests from going to the ACL. Chang et al. use a Bloom filter as a cache before sending the packets to the ACL [17]. Chvets et al. proposed a cache architecture which takes advantage of the type of locality that is present in the network packets [18]. Li et al. examined various packet classification cache designs with different associativities, hash functions and replacement policies to detect the effect on the performance [19].

Typical custom hardware approaches utilize Ternary CAM-s (TCAM) for packet classification purpose [20]. TCAMs store the rules in decreasing priority order and leverage "don't care" bits to enable ACL-like rules matching in hardware. All the rows in the TCAM are matched in parallel with the incoming key. At the output a priority encoder ensures that the highest priority match is the output match. TCAMs are fast and simple, however, there are a few drawbacks to TCAMs:

- TCAMs have very low storage density because of the following reasons:
  - One bit in TCAM requires 14-16 transistors [20, 21, 22].
  - Range specifications need to be split up into multiple entries.
- Limited scalability to long input keys due to the usage of exhaustive search approach.
- TCAM is much more expensive than SRAM [12].
- TCAM-s are very power hungry due to two main reasons:
  - The parallel comparisons in TCAM give rise to huge capacitive loads, resulting in access times three times longer than SRAM [23].

– Power consumption per bit of storage is around 150 times more than SRAM [23].

We propose a software firewall which uses a probabilistic data structure as a preclassifier and a cache type data structure after that to filter even more of the requests before they reach the ACL. The ACL lookup is done only in case the packet is absent in both the preclassifier and the cache. The critical path does not go through the ACL and is fast. The non critical path checks the ACL and updates the required data structure.

CHAPTER III

BACKGROUND AND MOTIVATION

Figure III.1 shows an example of a small network including a firewall. There are three receiver devices *Dev*1, *Dev*2 and *Dev*3. There are two hosts - *Good host* and *Malicious host*. Packets from *Good host* are good packets and should not be stopped from reaching the devices. Packets from the *Malicious host* are harmful packets and should be discarded before they can reach the devices. To protect the devices, a *Firewall* is placed just between the Internet and the devices in the network.

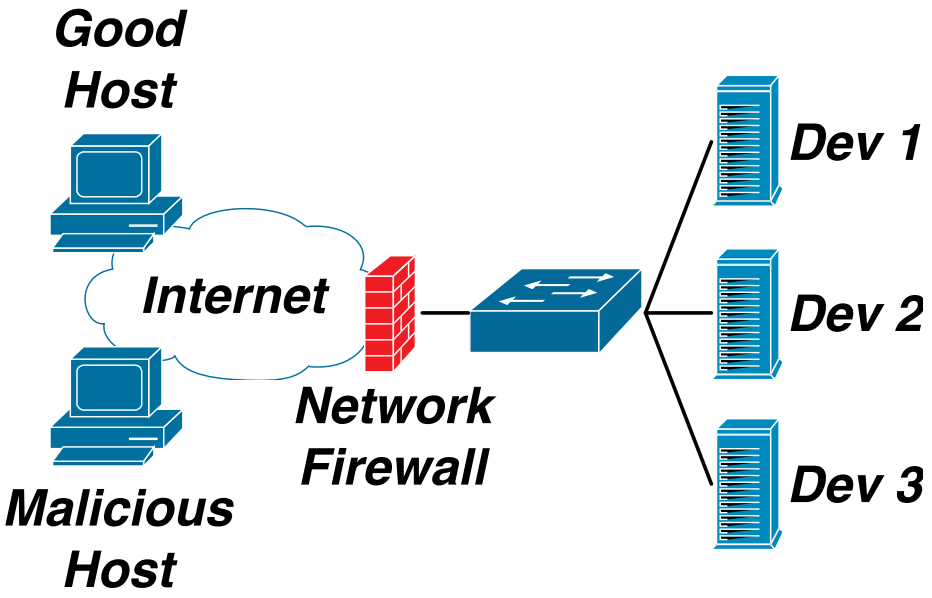


Fig. III.1. An example of a small network including a firewall

III-A. Access Control Lists (ACLs)

The firewall in Figure III.1 protects $Dev1$, $Dev2$ and $Dev3$ from harmful packet flows from the Internet. Firewalls are configured according to a set of rules called the *Access Control List* or *ACL*. A rule in an ACL is of the following form:

$$predicate \rightarrow decision \tag{3.1}$$

The *predicate* is a boolean expression over the different fields in the packet header, which are - source address, destination address, source port, destination port and the packet type. *Decision* can refer to either "a" for *Allow* or "d" for *Deny*. The packet is matched against the rules and if the *predicate* of a rule matches the packet, the *decision* of that rule is applied.

Rule no.	Src address	Dest address	Src Port	Dest Port	Type	Predicate	Decision
r_0	Any	Dev1	Any	25	Any	(Dest==*Dev1*) & (Dport==*25*)	Allow
r_1	Good Host	Dev2	Any	Any	Any	(Src==*Good Host*) & (Dest==*Dev2*)	Allow
r_2	Malicious Host	Any	Any	Any	Any	(Src==*Malicious Host*)	Deny

Table III.1. Firewall rules for the network in Figure III.1

Rule no.	Src address	Dest address	Src Port	Dest Port	Type	Predicate	Decision
r_1	Good Host	Dev2	Any	Any	Any	(Src==*Good Host*) & (Dest==*Dev2*)	Allow
r_2	Malicious Host	Any	Any	Any	Any	(Src==*Malicious Host*)	Deny
r_0	Any	Dev1	Any	25	Any	(Dest==*Dev1*) & (Dport==*25*)	Allow

Table III.2. Modified order of firewall rules

Table III.1 shows an example of an ACL for the network shown in Figure III.1. The 1^{st} column gives the rule number - there are 3 rules in this ACL. The next 5 columns - source address, destination address, source port, destination port and type make up the predicate. The predicate is shown in the next column. The fields in the header which are non-specific are not used in the predicate as they are don't care.

The decision is the last column - *Allow* or *Deny*. Rule r_0 in Table III.1 states that a packet from any source, to destination $Dev1$, from any source port, to destination port 25 will be *Allowed*. Rule r_1 states that a packet from *good host* going to $Dev2$ will be *allowed*. Rule r_2 states that any packet from source - *malicious host* should be *dropped*.

Let's take a simple case to explain a firewall's operation and a possible complication. Suppose a packet X arrives from *malicious host* for $Dev1$ to port 25. The packet is matched against firewall rules in Table III.1 to see if it should be *Allowed* or *Dropped*. X is for destination $Dev1$ and destination port is 25, which matches the predicate of rule r_0 as shown in Table III.1. Therefore Rule r_0 matches the packet. Rule r_1 does not match the packet as source for rule r_1 is *Good Host* but X is from the source *Malicious Host*. Rule r_2 again matches the packet as rule r_2 only mentions that the source should be *Malicious Host*. Thus X matches both the rules r_0 and r_2. But the rules are contradictory in their decisions - r_0 *Allows* the packet whereas r_1 *Denies* it. In such cases when there is a conflict, the first rule that the packet matches is the deciding rule. Because rule r_0 is the first matching rule and it *Allows* the packet, this packet will go through the firewall.

The ACL in this case is not designed well as a packet from the *malicious* host was allowed into the protected network. This shows that the order in which the rules are setup will have a great impact on the functionality of the firewall.

If the order of the rules is changed to the one as shows in Table III.2 by inserting r_0 from Table III.1 at the end in Table III.2 , and pushing up r_1 and r_2, then the scenario becomes quite different. Considering the same example of packet X from *malicious* host, again rules r_2 and r_0 match the packet but in this case the topmost matching rule is r_2 which *Denies* the packet. Thus the final decision of the firewall

would be to drop the packet.

This example shows there is a linear dependency introduced among the rules of the firewall. The fact that the i^{th} rule matches a packet, does not have any value, until we know that rules 0 to $i-1$ did not match the packet. This puts a constraint on the search algorithm of the ACL. Just finding a match would not be the end of the algorithm. A naive way to search the ACL is a simple linear search. This algorithm checks each rule before it checks the next rule. If there are N rules in the ACL, the worst case time taken for the search will be of the order O(N). Previous research regarding the search algorithm has been discussed in Chapter II.

Generally there is a single ACL for all the rules, good and bad packets go through the same path. If there are a lot of new packets waiting to go through the firewall, a packet X needs to wait for a decision to be made for each packet in queue previous to X. This happens irrespective of whether X is a *good* or a *bad* packet and whether the previous packets were *good* or *bad*.

Let us go back to the network in Figure III.1. Let the ACL be set up as in Table III.2. The $Malicious\ Host$ wants to attack the firewall, and it sends a flood of packets. Each packet is first compared against r_0, which does not match because the source is different. It is then checked against r_1 and the packet is dropped. But due to the high volume of packets sent by the malicious host, more number of packets arrive in a time interval than the ACL table is able to process. This causes the input queue to the ACL table to get filled up. Suppose there is a new packet G from $Good\ Host$. G gets queued at the end of the queue, and it has to wait for all the *bad* packets before G in the queue to get *denied* before G is allowed to proceed forward. This is because the paths for allowed packets and denied packets are the same. If the total input bandwidth is higher than the speed at which the firewall is able to process the

ACL rules for each packet, the queues will continue to grow. Eventually, the queue data structures become too big for the system to maintain. When this occurs, typical firewalls begin dropping packets, arbitrarily, without first processing them through the ACL. At this point, packets from *Good Hosts* may be dropped along with those from the *Malicious Hosts*, a form of Denial of Service (DoS) attack. Thus, firewall performance, i.e. the speed at which the ACL can be processed, is critical to the functionality of the firewall.

III-B. Bloom Filters

Our approach speeds ACL processing by removing the ACL from the main critical path, substituting it with a stochastic data structure as a pre-classifier for the packets. For this purpose, we use a Bloom filter [24].

A Bloom filter is a data structure which can be used to check if an element is present in a set or not. It was invented by Burton Bloom in 1970. Searching the Bloom filter can be accomplished in constant time, irrespective of the number of elements stored in it or the size of the Bloom filter. The Bloom filter utilizes very little space. The price paid for the memory and time efficiency is a probabilistic answer from the Bloom filter. The Bloom filter returns one of the two answers:

1. *Definitely* not present
2. *Probably* present

Though the Bloom filter can give a definite negative, the positive is probabilistic. There is a small probability that the *Present* answer can be false. This is why Bloom filters are said to have *false positives* but can never have *false negatives*.

The Bloom filter is made up of an array of bits or a bit vector. Initially all the

bits are set to 0. To store an element say A, some bits at indices, calculated using a function $f(A)$, are set to 1. To check for an element B, the bits at indices $f(B)$ are checked. If any of the bits is 0, B is $definitely$ not present. If all the bits are set to 1, then the element B *might* be present in the Bloom filter.

We can see that the search time taken is constant comprising the time taken to compute the indices using the function $f(x)$, and then the time taken to look up the actual bits at those indices. The same thing applies to updating Bloom filter.

The function used to compute the index is generally a hash function. This function takes the input and converts into a different random number. The simplest hash function is built by xor-ing bytes of the data together to get a smaller compact value. The quality of the hash function is measured by how well the output values are randomized over the entire output range. The drawback of a hash function is that, since we lose some information during the hash, more than one input value might give the same output value, thus giving rise to *aliasing*. This is the cause for having a probabilistic answer from the Bloom filter.

A Bloom filter generally uses more than one hash function. This means that each element gives rise to more than one index in the Bloom filter. This helps to reduce the effect of aliasing as the chances of all indices for a particular element being aliased with indices of different elements would be significantly lower.

While inserting an element into the Bloom filter, all the hash functions (say k) are computed using that particular element to get k indices. These k indices are set to 1. While checking for an element, k indices are again computed, and the values at these indices are checked. As explained earlier, the k values have to be 1 for a possibility of the element being present in the Bloom filter.

The false probability of Bloom filters can be calculated analytically [25]. After

inserting n keys into the bloomfilter of size m, the probability that a bit is still 0 is

$$\left(1 - \frac{1}{m}\right)^{kn} \tag{3.2}$$

The probability of false probability in this situation is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx (1 - e^{kn/m})^{k} \tag{3.3}$$

The RHS is minimized in the condition

$$k = ln(2)\frac{m}{n} \tag{3.4}$$

False positive rates for a few combinations of m to n ratios and k are given in Table III.3. From the table it can be seen that the false probability rate decreases with increase in k for a particular m/n ratio, but after a minimum value it increases again. For example, when $m/n = 8$ the optimal value of k is 5.55 and the false probability rate is minimum for $k = 5$ and $k = 6$. For a particular value of k though, the false probability rate decreases with increase in m/n.

m/n	k	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8
2	1.39	0.393	0.400						
3	2.08	0.283	0.237	0.253					
4	2.77	0.221	0.155	0.147	0.160				
5	3.46	0.181	0.109	0.092	0.092	0.101			
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578	0.0638		
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347	0.0364		
8	5.55	0.118	0.0489	0.0306	0.024	0.0217	0.0216	0.0229	
9	6.24	0.105	0.0397	0.0228	0.0166	0.0141	0.0133	0.0135	0.0145
10	6.93	0.0952	0.0329	0.0174	0.0118	0.00943	0.00844	0.00819	0.00846

Table III.3. False probability rates in a bloomfilter

Bloom Filters have been used in various network applications. For example, Bloom Filters were used in resource routing for resource discovery service [26] or in routing a file request efficiently in a peer to peer network to get a probabilistic routing

algorithm [27]. They have been used in packet routing to detect forwarding loops early [28] or for multicast routing [29]. A survey of different network applications can be found in [30]. We present the first work we are aware of, which utilizes a Bloom filter in a high performance software firewall for commodity hardware.

CHAPTER IV

OUR APPROACH

In this thesis we propose an architecture for the firewall to fulfill the following two properties:

- The paths for allowed packets and denied packets are separated.
- The ACL lookup is removed from the critical path, ACL lookup is used for *learning*, but for known packets the ACL lookup is bypassed.

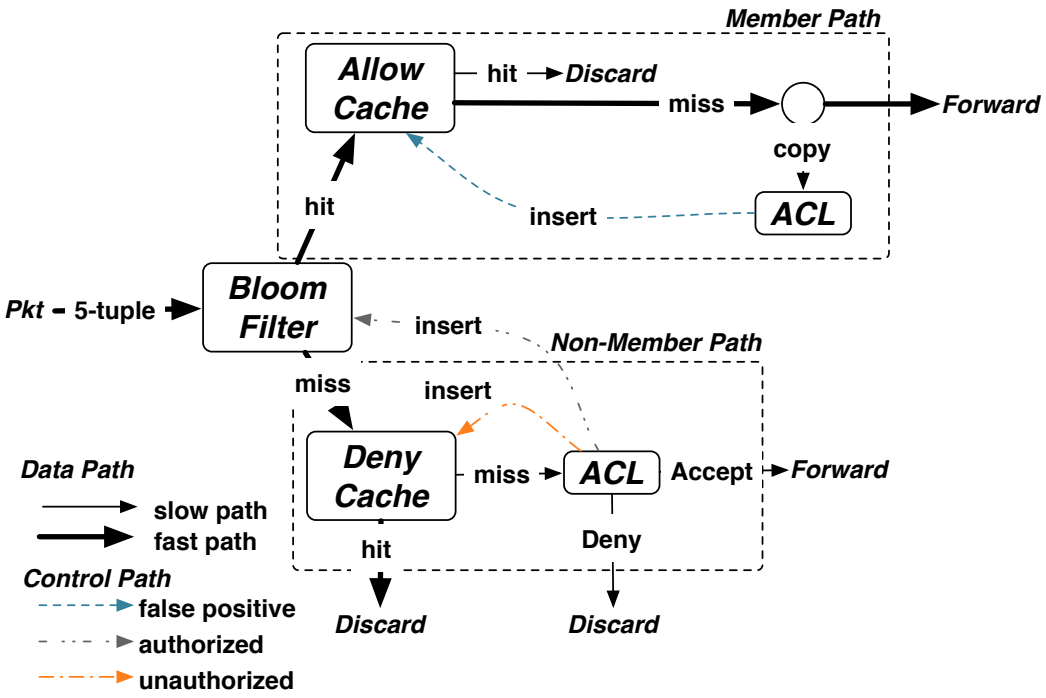


Fig. IV.1. Proposed design of the firewall

These two properties help to protect against Denial of Service attacks and also speed up the functioning of the firewall.

The proposed architecture is shown in Figure IV.1. It consists of the following parts, they are described in detail below after the description of the overall architecture.

- Bloom Filter
- Cache
- ACL

IV-A. Overall Architecture

Algorithm 1 corresponds to Figure IV.1. The incoming packets flow through the Bloom filter and if they are present in the Bloom filter, they go through the *hit* path.

Algorithm 1 Firewall Algorithm

index ←skb_index(*skb*)
bloomfilter_verdict ←check_bloomfilter(*index*)
if *bloomfilter_verdict* = 1 **then**
    bloomfilter_present_path(*index*,*skb*)
else
    bloomfilter_absent_path(*index*,*skb*)
end if

Algorithm 2 Bloomfilter Present Path

```
procedure BLOOMFILTER_PRESENT_PATH(index,skb)
    allow_verdict ←check_allow_cache(index)
    if allow_verdict = 1 then
        Drop Packet
    else
        if Packet should be sampled then
            iptable_verdict ←check_iptable(skb)
            if iptable_verdict = Deny then
                Update_allow_cache(index)
                Drop Packet
            else
                Allow Packet
            end if
        else
            Allow Packet
        end if
    end if
end procedure
```

The *hit* path of the algorithm is shown in Algorithm 2. These packets then go through the *Allow cache*, the presence in which (*hit*) denotes that the packet is false positive and needs to be dropped. If the packet is not present, it is allowed into the system by the *miss* path. Even if the packet is allowed, a few of the packets are sent to the ACL for reclassification. In the event reclassification shows the packet should have been allowed, nothing is done. If, however, the packet should have been denied,

the *Allow Cache* is updated through the *insert* path.

Algorithm 3 Bloomfilter Absent Path

```
procedure BLOOMFILTER_ABSENT_PATH(index,skb)
    deny_verdict ←check_deny_cache(index)
    if deny_verdict = 1 then
        Drop Packet
    else
        iptable_verdict ←check_iptable(skb)
        if iptable_verdict = Allow then
            Update_bloomfilter(index)
            Allow Packet
        else
            Update_deny_cache(index)
            Deny Packet
        end if
    end if
end procedure
```

As shown in Algorithm 3, if the packet is not present in the Bloom filter, it goes on the *miss* path. The packet is matched in the *Deny Cache* to see if it should be discarded via the *hit* path as a known bad packet. In case it does not match any of the previous cases, then it is a new packet, and the ACL is checked. Depending on the output of the ACL, if the packet is a good packet the bloom filter is updated and the packet is allowed into the system. If the packet turns out to be a bad packet after checking the ACL, the *Deny Cache* is updated and the packet is dropped.

IV-B. Bloom Filter

In our proposed architecture, we use the Bloom filter to initially divide the traffic into two paths - likely allowed and likely denied. The Bloom filter contains the packets which are known good packets, and if the packet matches in the Bloom filter it goes to the *hit* path. If the packet does not match in the Bloom filter, it goes to the *miss* path.

Byte#	Description	Entropy
0	Source byte 1	3.809915
1	Source byte 2	4.767318
2	Source byte 3	5.024942
3	Source byte 4	5.465738
4	Destination byte 1	3.945233
5	Destination byte 2	4.552347
6	Destination byte 3	5.026454
7	Destination byte 4	5.468756
8	Source port byte 1	3.722121
9	Source port byte 2	3.815076
10	Dest port byte 1	4.264202
11	Dest port byte 2	4.470557
12	Protocol	0.444688

Table IV.1. Entropy of each byte in a sample trace

In our implementation, we used a Bloom filter with 65536 entries. The index into the Bloom filter is 16 bits. We used five different hash functions to index into

Byte#	Description	Entropy
7	Destination byte 4	5.468756
3	Source byte 4	5.465738
6	Destination byte 3	5.026454
2	Source byte 3	5.024942
1	Source byte 2	4.767318
5	Destination byte 2	4.552347
11	Dest port byte 2	4.470557
10	Dest port byte 1	4.264202
4	Destination byte 1	3.945233
9	Source port byte 2	3.815076
0	Source byte 1	3.809915
8	Source port byte 1	3.722121
12	Protocol	0.444688

Table IV.2. Entropy of each byte in a sample trace in descending order

the filter. When designing the hash functions, we first measured byte entropy of all the bytes in the header fields in a network trace using the following equation

$$entropy(x) = -ln(prob(x)) * prob(x) \tag{4.1}$$

The byte entropies of the sample trace is shown in Table IV.1 and they are arranged in descending order in Table IV.2. The entropy values match the expectations. The lower two bytes of the source address and the destination address have a higher value of entropy than the higher bytes which are set for the organization. Entropy

values for the ports are in between. The protocol field has the least entropy again as expected, as it can hold only very few specific values like TCP, UDP, $ICMP$ etc.

To have a high degree of entropy in our hash functions, we ensure that the two highest entropy bytes are not XORed together when making our hash index. To this end, we combined the byte with the most entropy with the byte with third highest entropy and so on to get the first byte in the index. For the second byte in the index, we started with the byte with the second highest entropy and combined with the fourth highest entropy and so on as shown in the following equations. We have not used the protocol field in our calculation but it should be used as the protocol is a deciding field in a lot of the rules. Therefore it will be added in the future work.

$$\begin{aligned} byte1 &= entropy(byte7) * entropy(byte6) * entropy(byte1) \\ &\quad * entropy(byte11) * entropy(byte4) * entropy(byte0) \end{aligned} \tag{4.2}$$

$$\begin{aligned} byte2 &= entropy(byte3) * entropy(byte2) * entropy(byte5) \\ &\quad * entropy(byte10) * entropy(byte9) * entropy(byte8) \end{aligned} \tag{4.3}$$

After getting the different indices from one trace, we plotted the probability distribution function (pdf) of indices shown in Figure IV.2. As can be seen, the pdf of the index is almost flat with no very prominent spikes. While higher entropy could be maintained by ensuring that the 16 highest entropy bits in the 5-tuple were not XORed with each other, we find that this technique would require substantially higher complexity, lowering performance without reducing collisions significantly.

This baseline index value was then XORed with five random values to come up

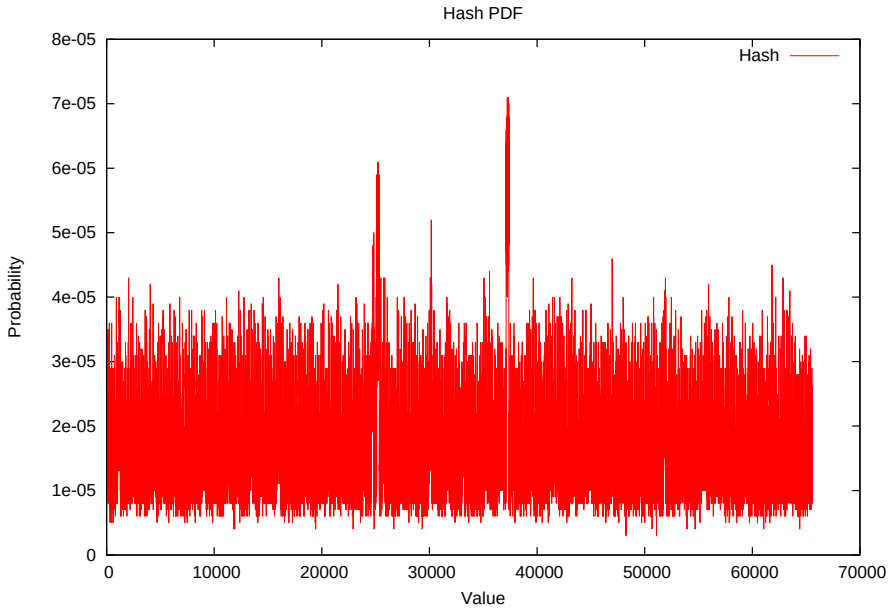


Fig. IV.2. Probability distribution function for hash function

with the five hash functions needed by the Bloom filter.

As the bloomfilter gets filled by adding more and more elements, the percentage of false positive goes higher. Therefore the bloomfilter needs to be *aged* gracefully such that the percentage of false positives stay within a fixed limit. Two methods have been implemented for aging the bloomfilter.

- *Cold Clear* - The bloomfilter is cleared completely after a certain number of elements have been added to it.

  This has the drawback that the simulation starts from a clean slate after every cycle.

- *Random bit clear* - After a certain number of elements have been added to the bloomfilter, for every additional update a few random bits are cleared. This

keeps the number of bits set in the bloomfilter at an approximately constant level.

In this case, the bloomfilter never gets completely cleared and always has a number of elements in it. The disadvantage is that the bloomfilter might get polluted, that is some of the elements which had been added would not be present any more because of the clearing of one or more bits. If the pollution increases a lot, it will affect performance where most of the elements do not hit in the bloomfilter because at least one of their bits have been set to 0.

As we saw in Table III.3, with $k = 5$ and $m/n = 10$ the false probability rate should be around 0.00943. We note, because the indexes are XORed with random numbers which are regenerated once every hour, a prospective attacker must guess all five random numbers in order to produce a packet which would be guaranteed to produce a false positive. Furthermore, as we will discuss in the next section, these false positive packets are further filtered by the allow cache, so only the first packet of a false-positive flow would be allowed to penetrate the firewall.

IV-C. Cache

We use a cache like data structure in our design, which helps us to filter out some of the packets before sending to the actual ACL table. Generally, a cache is a small, fast storage structure which maintains data that has high temporal locality. In a firewall implementation, caches can keep a few of the latest used rules for future packets to be used directly instead of going through the ACL again. This helps in speeding up the path of the packets through the firewall or the packet filter.

In our architecture, we use two caches - one on the allow path and one on the deny path. Both of them store packets which are *known bad* packets. Therefore if

any of the caches contain the incoming packet, the packet is dropped instantly and not processed any further. On the allowed path, the cache stores the packets which are *false positives* that is they should not be allowed into the system but because of the probabilistic nature of the Bloom filter they are allowed. The cache on the deny path functions as a simple cache so that for every packet the main table does not need to be looked up, if the packet has been seen before and is a known bad packet, it can be dropped instantly.

Our architecture utilizes a Bloom fliter preclassifier as well as two caches. Naïvely one might design an architecture with a single cache without a preclassifier stage. If only a single cache is used, it would only store a small subset of the last few packets and the decisions. This would include *allow* decisions as well as *deny* decisions. In our proposed architecture, the Bloom filter preclassifies the packets by storing only the *allow* packets. The cache on the *hit* path stores only the $false\ positive$ packets, therefore the time taken to search through this cache is relatively very less. The cache space is thus much more efficiently utilised due to the presence of the Bloom filter. Even the cache on the *miss* path stores only the packets which should be dropped, therefore decreasing the cache size requirement. The sizes of the caches used are shown in Table IV.3.

Our cache design is direct mapped but the cache is indexed by hashing the incoming data. The cache is hashed with the base hash value that was combined with random numbers in the case of the Bloom filter. The *Allow Cache* can be made much smaller than the *Deny Cache*.

	Allow Cache	Deny Cache
Index size	8 bit	16 bit
Entries	256	65536
Size(bytes)	8k	2M

Table IV.3. Cache specifications

IV-D. ACL

The ACL in our architecture is referenced when we receive a new packet for the first time, that is, it does not exist in either the Bloom filter or the cache on the denied path. Depending on the outcome of the ACL, the Bloom filter or the denied cache is updated. We also use the ACL at some regular intervals in the allowed path to detect false positives. The outcome of the ACL in this case is used only if it is negative and the cache on the allowed path is updated.

In our implementation we use the default ACL present in the linux kernel - the *xtable* and the user implementable part of it - *iptable*. The setup consists of chains to which rules are added. There are five predefined chains:

- *Prerouting* - Packets enter this chain before routing decisions are made.
- *Input* - Packets enter this chain if they are meant for this cpu.
- *Forward* - Packets enter this chain if they have to be forwarded on to a different network.
- *Output* - This chain is for outgoing packets.
- *Postrouting* - This chain is entered just before handing the packets to the

hardware.

Rules can be added to each chain, and these chains are accessed via netfilter hooks from within the linux kernel networking stack. User defined chains can be added which can be reached from one of these predefined chains by adding a rule to jump to the beginning of the user defined chain. A rule in the chain can also have a decision like *Accept* or *Deny*. Each packet goes through each chain in turn until it reaches a decision by a matching rule, or it reaches the end of the chain or it is rerouted to another chain. When the end of the user defined chain is reached, the packet comes back to the original location in the chain it was traversing earlier.

We add all our rules to the *Forward* chain to set up the ACL.

IV-E. Stateful Firewall

The firewall that has been implemented is a stateless firewall, but it can be converted into a stateful firewall with a few additions. A stateful firewall is one which tracks outgoing requests from the trusted interface and modifies the behaviour of the firewall. This allows responses through the firewall for outgoing requests from the trusted interface. For implementing statefulness, rules should be added to the ACL once there is a outgoing packet from the trusted interface. That rule would be the complement of the outgoing packet, that is the source in the rule should be the destination of the outgoing packet and vice-versa. This rule will be deleted after a specific amount of time of no activity related to that rule. This prevents malicious packets being accepted after the response is received.

The addition of the rule does not pose any problem as the bloomfilter gets trained once the rule is in the ACL. When the rule is removed though, the bloomfilter and the deny cache needs to be cleared. This is needed so that packets from that source

is not allowed in after the rule has expired in the ACL.

CHAPTER V

METHODOLOGY

We implemented our proposed firewall architecture as a kernel module. The kernel module receives incoming packets from the network interface card - $eth0$ and passes it through the firewall to the linux networking stack as shown in Figure V.1.

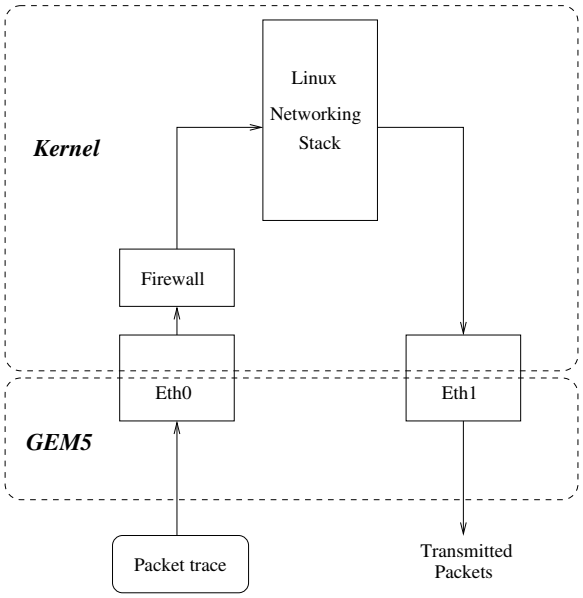


Fig. V.1. Block diagram for simulation setup

The linux kernel that is used is version 2.6.27, running on the GEM5 multicore processor simulator [31]. This simulator simulates a full system including network interface cards (NIC) and IO connections. Two network interfaces are instantiated in GEM5 in the hardware part of the simulator as shown in Figure V.1. These interfaces have a hardware component as well as a software device driver component. One NIC is used as the trusted interface, which points towards the private network ($eth1$) and the other one is the untrusted interface, that is the interface pointing towards the rest

of the network ($eth0$). The traffic comes in through the untrusted interface $eth0$. The firewall, which is instantiated as a virtual device driver, grabs the packets and passes them through the firewall. Only the packets which are allowed by the firewall are sent on to the ip routines in the Linux Networking Stack. The denied packets are just dropped here and no further actions are taken for them. The performance metrics are observed in the GEM5 portion of $eth1$ for the packets that are transmitted out of the system.

The details of the system setup for GEM5 is given in Table V.1.

To evaluate the performance of our design, we need a network trace to feed into the simulator. For this purpose, we used anonymized traces from CAIDA servers [32].

For generating the iptable, we fix a percentage of unique traffic from the traces as bad traffic. We have 10, 20 and 30% of bad traffic and generate an iptable from this which can be used in the kernel using the command 'iptable_restore'. from the bad traffic flows. There is a correlation between the number of rules in the iptable and the percentage of bad traffic which is shown in Table V.2. The number of rules increases with the increase in the bad percentage of traffic.

We could also make an iptable with some common rules like *Drop all udp packets* or *allow all packets to port 80*. This would make the iptable more similar to the commonly used iptables. Also we had each rule explicitly mentioning the ip addresses. Instead of this, multiple rules could be condensed to make a single rule, thus decreasing the number of rules in the iptable and increasing the processing speed of the firewall. We will work on these on the ongoing project.

The performance of the proposed firewall architectre is measured against a baseline case. The simulation topology remains the same even for the baseline case, the only difference being that in the baseline case, the firewall comprises only the iptable.

Name	Value
Architecture	Alpha
CPU model	Out of order
Frequency	4GHz
L1 ICache	32kB
L1 DCache	64kB
L2 Cache	2MB
Bus Frequency	4GHz
IO Bus Frequency	4GHz
Memory Model	Simple
Kernel version	2.6.27
NIC Card	IGbE_e1000(i8254x)
Driver version	7.3.21-k3-NAPI
Rx Desc Ring	4096
Tx Desc Ring	256

Table V.1. Setup for simulation

A packet trace is used as the source of incoming packets at $eth0$. In the original kernel setup, the iptable is invoked only from the ip routines which is at a higher layer in the networking stack than the location of our firewall. For this purpose we needed to modify the linux kernel slightly. The netfilter hooks which had been earlier embedded in the networking stack have been commented out as they are called separately as an iptable call for the baseline case as well as the proposed case. The modification was done even for the baseline case to prevent the two simulation setups being different.

Percentage of Bad traffic	Rules in iptable
10	2678
20	5357
30	8928

Table V.2. Number of rules in iptable for percentage of bad traffic

For performance evaluation, we measure the following metrics for both the aging mechanisms:

- Latency per packet
- Bandwidth
- Number of dropped packets
- False positive rate

We read the trace file and inject the packets into the system through the untrusted NIC according to the timestamps in the traces. While injecting the packets we add the insertion time into the packet data. When the packet is ejected out of the system we measure the difference between the ejection time and the injection time saved in the packet buffer. We calculate an average of this difference over a number of packets. We do not take into account the packets which were dropped by the firewall. The bandwidth is also measured here. In case the packet was discarded in the receiving stage due to memory buffers being full, it is counted as a dropped packet. The dropped packet numbers are also reported to show a reduction in this number from the baseline to the proposed case.

The simulation outputs an unique id per packet for all the packets that are forwarded through the firewall. This list of unique id-s is matched with the iptable using a script to count the number of bad packets which should not have been allowed into the system but are actually being allowed. This number gives the false positive rate.

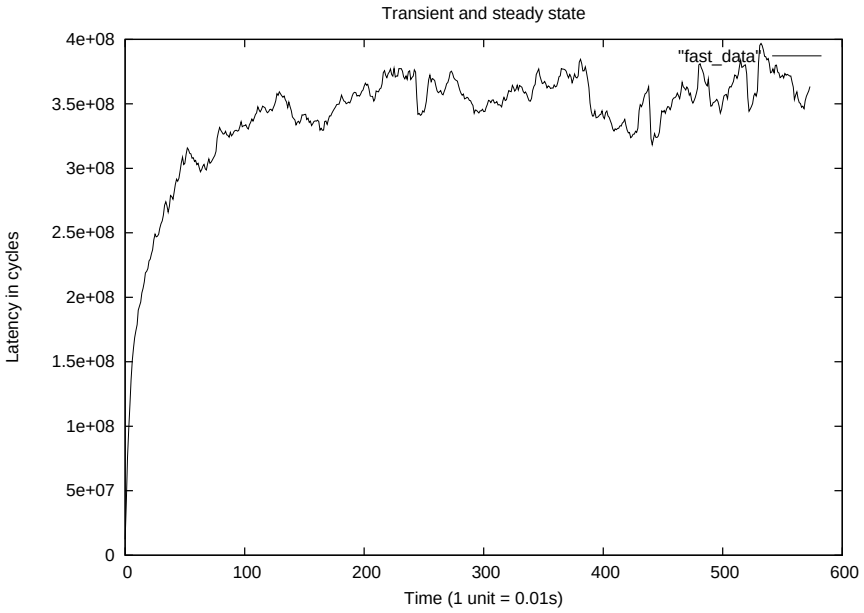


Fig. V.2. Transient to steady state latency per packet

The latency is reported in the number of cycles taken by a packet from the time it entered the system through an ingress network interface till the time it was forwarded through an egress network. The latency and bandwidth are only measured in the steady state after a warm up phase, as shown in Figure V.2. This is because for the initial packets, the latency per packet will be lower as can be seen from the graph. If the packets come at a steady rate which is higher than the rate at which

packets are being processed, the following packets would have to wait in queue till the previous packets have been processed. Therefore the initial packets go through the system at a faster rate whereas the following packets have a higher latency due to the added waiting period. In the steady state, all the packets have the waiting period before they are processed, and therefore the latency remains the same from this time onwards as long as the steady input rate is maintained.

We can also measure the latency for each part of the code, i.e. the time taken by the bloomfilter, the time taken by the caches, and the time taken by the ACL. This we would do in future work.

We run the simulation for $150,000$ packets, among which it is made sure that steady state was achieved for all cases within the first $50,000$ packets. The latency and bandwidth are measured in the next $100,000$ packets. The number of dropped packets and the false positive rate are measured over all the $150,000$ packets.

Since we use Alpha architecture and kernel version 2.6.27, we cannot keep up with a line speed ($10Gbps$) for the baseline or the fastest case in our proposed approach. 2.6.27 version is old, and is not optimized for Alpha architecture as well. At this speed almost all the packets get dropped due to a huge difference in the receiving bandwidth and the transmission bandwidth. For this reason we inject packets into the system at the rate of $1Gbps$.

We have packets dropped even at the lower speed of $1Gbps$, and we could run simulations at such a low speed as to not drop any packets. We have maintained a higher speed for simulating a more realistic scenario where packets get dropped in steady state. In further works we will also simulate a slower scenario.

CHAPTER VI

RESULTS

We present the results for the Incremental cleared bloom filter first followed by the results for the Cold cleared bloom filter case. After that we compare the two cases.

VI-A. Incremental Cleared Bloom Filter

First we report the results for the *Incremental Clear* case. In this case, after a cutoff in the number of elements present in the bloom filter is reached, the occupancy is maintained at the same level by clearing bits. This cutoff is also referred to as *Clearing interval* in the rest of the chapter.

Next we analyze each of the performance metrics in turn - latency, bandwidth, reduction in dropped packets and the false positive rate.

VI-A.1. Latency

In Figure VI.1, the percentage improvement for latency in cycles per packets for the proposed system over the baseline system, is shown.

The improvement is negative in two cases

- 10% bad traffic with a cutoff of 3000
- 10% bad traffic with a cutoff of 5000

In these cases, the time taken to check the bloom filter and the caches add up with the time taken to just check the ACL. These cases have a very low clearing cutoff as well as a low percentage of bad traffic. With this configuration most of the packets do not match in the bloom filter and go to the longer path which has the ACL check in place. Therefore the improvement is not seen in these cases.

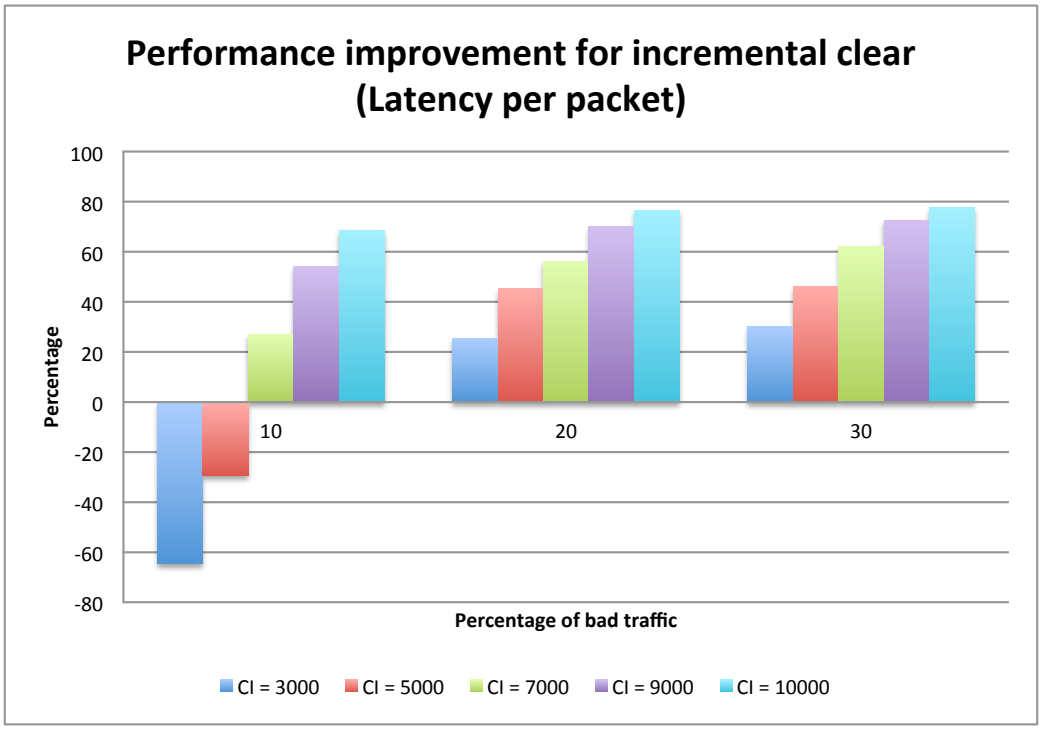


Fig. VI.1. Percentage improvement in latency for varying percentages of bad traffic

The percentage improvement in latency decreases as the clearing cutoff decreases for the same bad traffic percentage. If the clearing cutoff is low, for each packets that is added to the bloom filter after that, some bits are cleared in the bloom filter. This adds an additional time to the packet processing time. This also pollutes the bloom filter to an extent, removing good entries. This causes the packets to follow the non-critical path through the ACL, thus increasing the average latency per packet.

The percentage improvement also increases with increase in the percentage of bad traffic. This is because the percentage of good traffic is highest in the case of 10% and to maintain a level of occupancy, we kick out good packets every time we add a packet into the bloom filter, which happens very often. This gives rise to false negatives. In the case of 30% we have a better working set already present in the bloom filter,

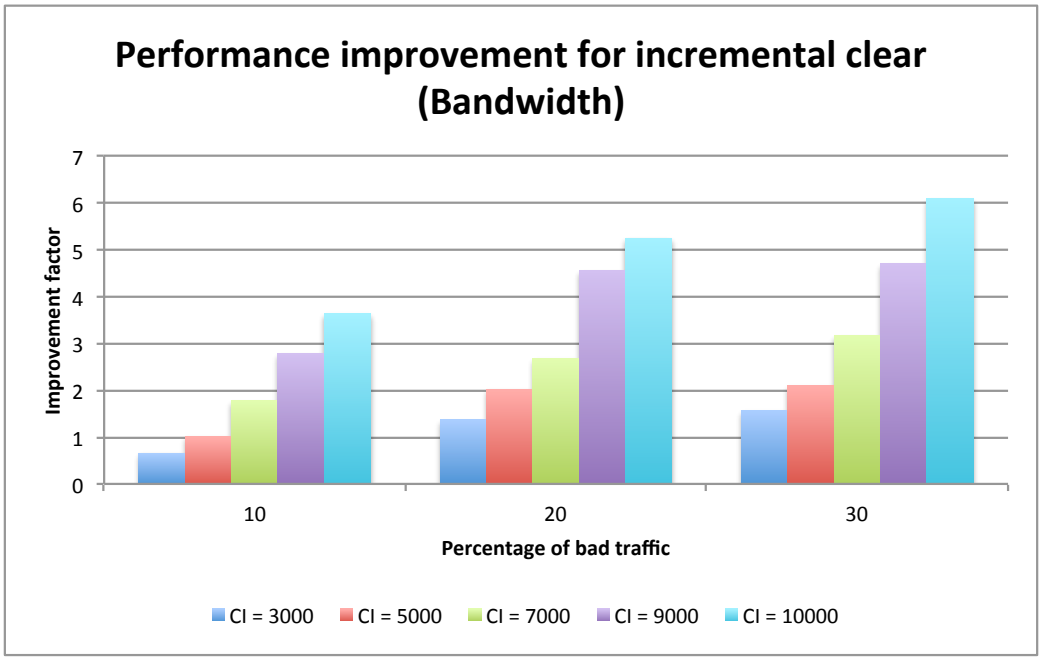


Fig. VI.2. Percentage improvement in bandwidth for varying percentages of bad traffic

and lesser number of packets are classified as good. This causes a reduction in the frequency of clearing bits from the bloom filter, reducing false negatives and therefore performance improves.

VI-A.2. Bandwidth

In Figure VI.2, the improvement for bandwidth for the proposed case is shown, as a factor of the baseline case. This is the transmission bandwidth which also reflects the bandwidth of packet processing in the packet firewalls.

The trend seen here is exactly the same as the latency shown in Figure VI.1. The improvement is in factors instead of a percentage, that is because the bandwidth is in bits per second which takes into account the size of the packets, not only the number of packets. The packet length extends to a maximum of 1500 bits.

For verification purposes, it can be seen that for the two cases that had negative improvement in latency, the normalized bandwidth has a value lower than 1, that is the bandwidth is lesser than the bandwidth seen for baseline.

VI-A.3. Reduction in Dropped Packets

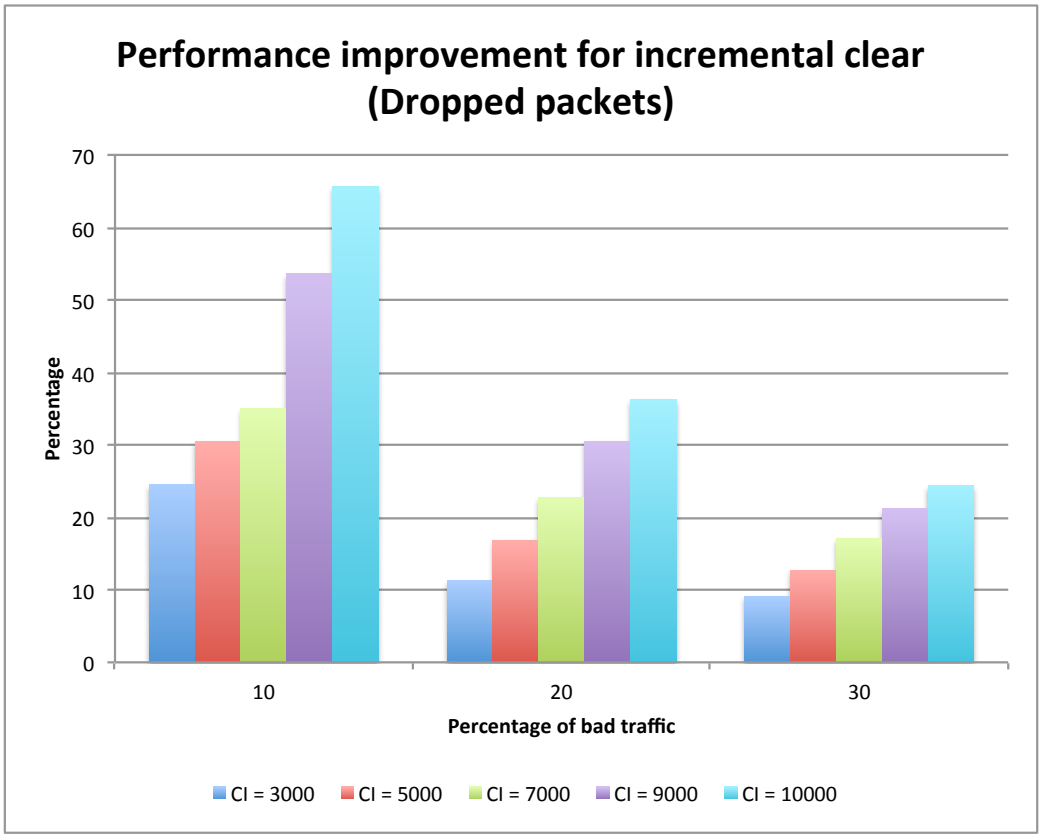


Fig. VI.3. Percentage improvement in reduction of dropped packets for varying percentages of bad traffic

Figure VI.3 shows the percentage reduction in the number of dropped packets. The percentage is calculated using the difference of the number of packets dropped in the proposed case and the baseline case over the number of packets dropped in the

baseline case.

Packets are dropped due to scarcity of resources. While the kernel processed the packets which have already arrived, the memory space and queues allocated for this process is full. This causes new incoming packets to be dropped due to lack of queue space.

For the case where the steady state latency had negative improvement also, there is a small improvement in the number of dropped packets. The latency for processing packets is lesser before the steady state is reached. Even though the bandwidth and latency are measured in the steady state, the number of dropped packets is measured throughout the simulation. Therefore in this case a small percentage improvement is seen even though the steady state latency and bandwidth show no improvement.

For a particular percentage of bad traffic, the percentage increases with higher clearing intervals. With higher intervals, the latency to process each packet reduces, and therefore the resources are emptied faster. This decreases the number of dropped packets.

With increase in the percentage of bad traffic however, the number of dropped packets increase. Even though the latency decreases for increasing bad traffic as shown in Figure VI.1, the latency does not take into account all the packets received by the system. The latency is calculated only for the packets which make it through, and even though the other packets get dropped, the resources that those packets use remain in use till they are dropped. With increase in percentage of bad traffic, the resources utilized by the bad traffic increases, which causes an increase in the number of dropped packets, reducing the percentage benefit.

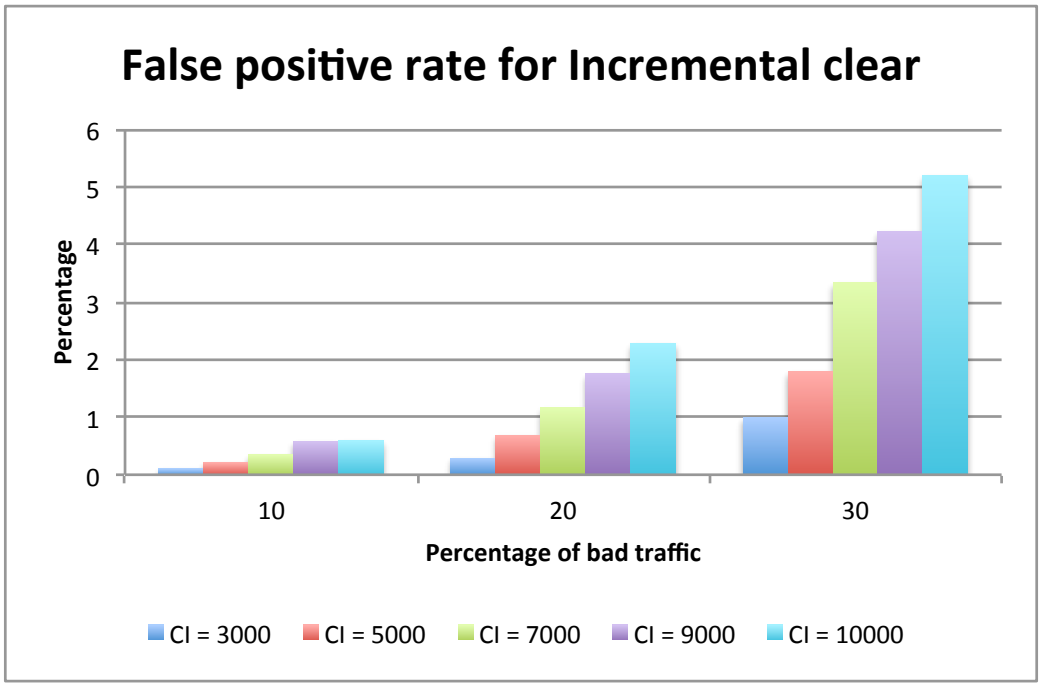


Fig. VI.4. False positive rates with different percentages of bad traffic

VI-A.4. False Positive Rate

Figure VI.4 shows the change in false positive rate with varying bad traffic percentage and clearing intervals. False positive rate is calculated as the percentage ratio of the number of bad packets which made it through the firewall to the number of total packets which made through the firewall.

The false positive rate is lower for lower clearing intervals. This is because the lower clearing cutoffs maintains the occupancy of the bloom filter to a very low number of elements. Lower occupancy reduces the number of false positives. But the false positive has a steep increase with increase in the clearing cutoff. That is because the occupancy is maintained at a higher value, therefore the chances of false positive increases.

With increase in the percentage of bad traffic as well, the percentages of false positive increases. This is because more number of packets have the probability to get classified as bad packets.

VI-B. Cold Cleared Bloom Filter

Now we discuss the simulation results for the cold cleared bloom filter case. In the cold clear, the bloom filter is totally cleared after a fixed interval, referred to as the *clearing interval*. This is the number of elements that are added to the bloom filter before the bloom filter is taken back to its initial state with all bits reset to 0.

VI-B.1. Latency

In Figure VI.5, the percentage improvement in latency for the cold cleared bloom filter setup over the baseline is shown. Similar to the Incremental clear case, the latency reduces for increasing clearing intervals. But the trend is not the same in the case of increasing percentage of bad traffic.

With increase in bad traffic, latency increases in this case. This is because number of good packets decrease, and it takes more time for the bloom filter to re-learn and get back to its useful state. This reduces the number of packets which pass through the bloom filter and more number of packets are rerouted through the ACL.

VI-B.2. Bandwidth

The normalized bandwidth is shown in Figure VI.6. The bandwidth follows the exact trend of latency as in the case of Incremental clear.

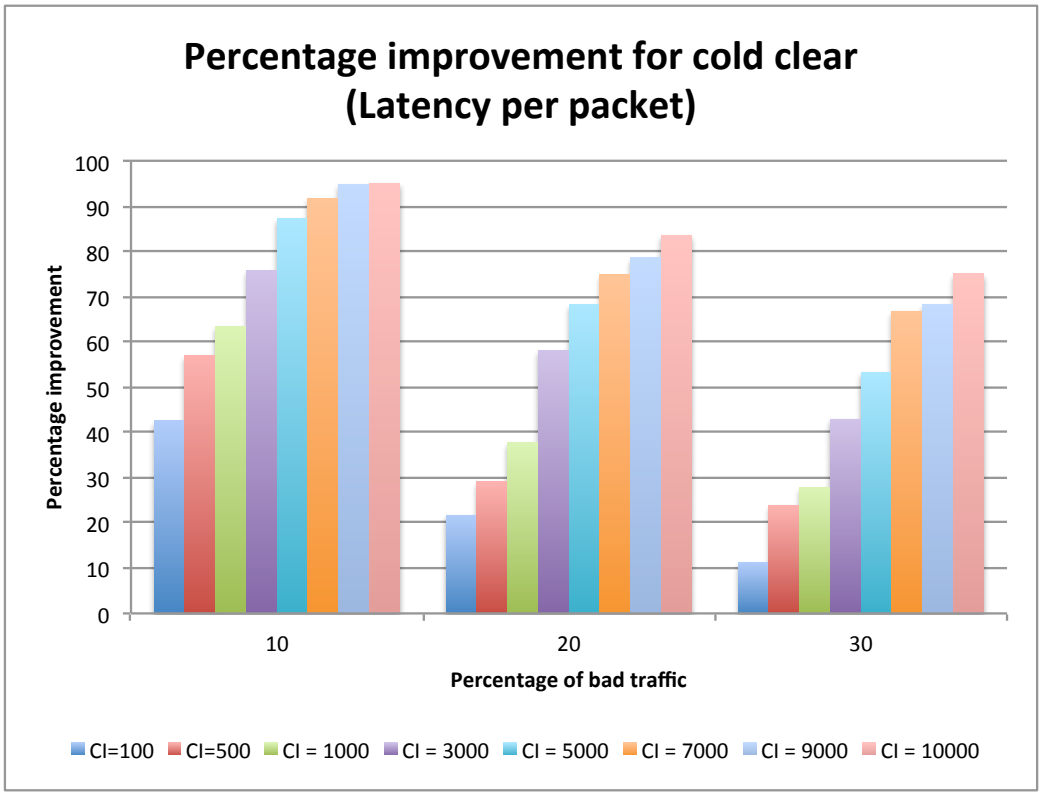


Fig. VI.5. Percentage improvement in latency for varying percentages of bad traffic

VI-B.3. Reduction in Dropped Packets

The percentage improvement in dropped packets is shown in Figure VI.7. The percentage improves with increasing clearing interval, as the latency reduction decreases the resource usage.

But the percentage of dropped packets reduces for increasing percentage of bad traffic. This is for a two-fold reason. Firstly, the latency improvement decreases, as shown in Figure VI.5, the increasing latency increases the time for which resources are utilized. The second reason is similar to the Incremental clear case, that is the number of bad packets, even though they do not affect the latency, do affect the

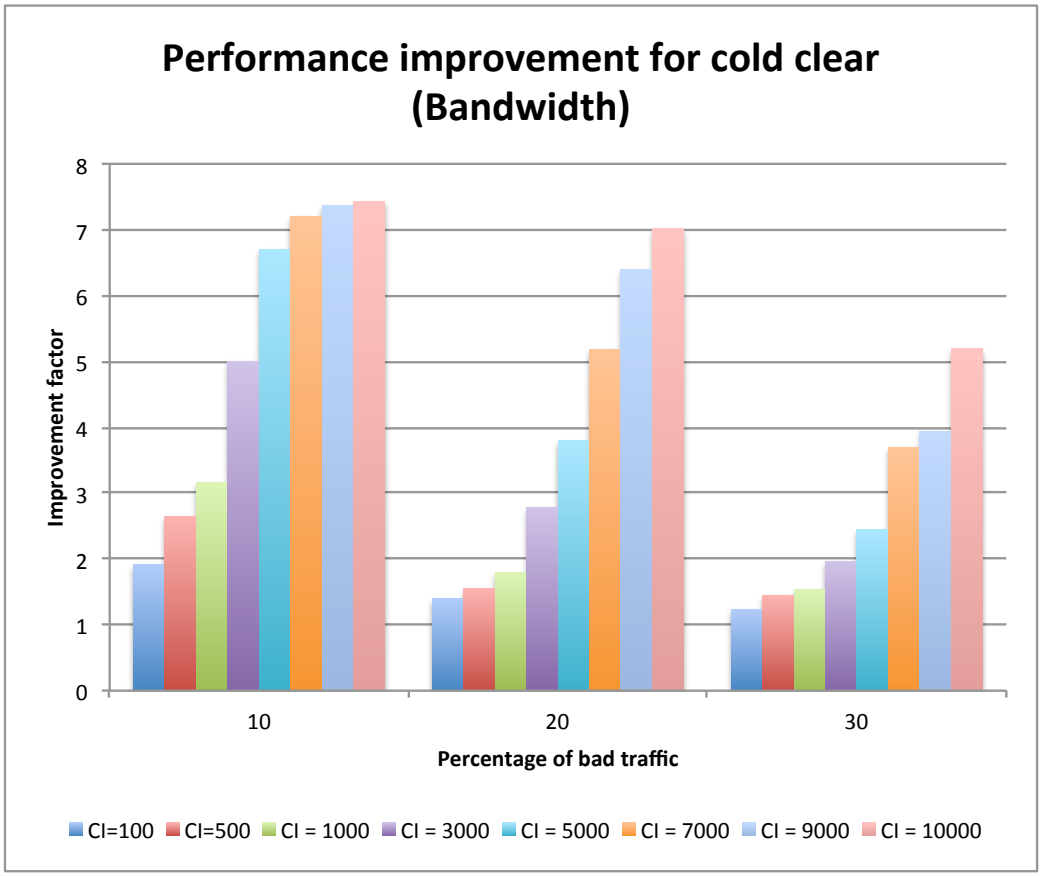


Fig. VI.6. Percentage improvement in bandwidth for varying percentages of bad traffic

resource usage. With increasing bad packets going through the ACL, which takes a longer time, the queues and memory spaces are not freed up sooner for new packets to utilize.

VI-B.4. False Positive Rate

The false positive rates for different percentages of bad traffic and clearing intervals are shown in Figure VI.8. The false positive rates are low in the case of lower clearing intervals, as the occupancy is very low before the bloom filter is cleared off again. As the occupancy increases, the false positive rate increases. With increase in bad

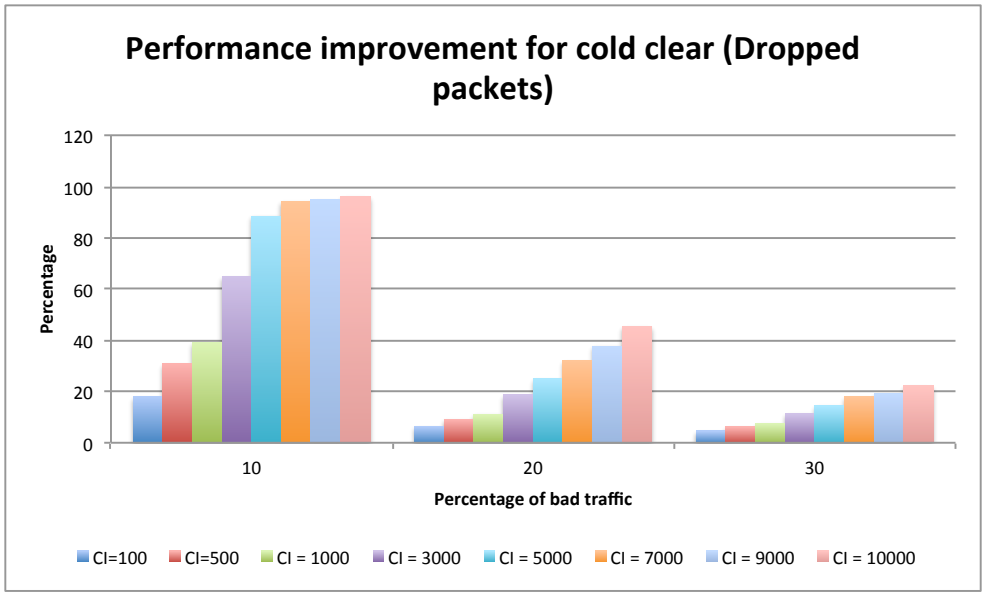


Fig. VI.7. Percentage improvement in reduction of dropped packets for varying percentages of bad traffic

traffic, the false positive rate increases. This is again because, with increase in bad traffic, the probability of a packet being classified as a bad packet increases.

VI-C. Comparison of Cold Clear and Incremental Clear Cases

Next we compare the cold clear bloom filter with the incremental clear bloom filter system. We compare the latency and the false positive rates. The bandwidth improvement follows a similar trend as shown in latency and the reduction in dropped packets is similar in both cases.

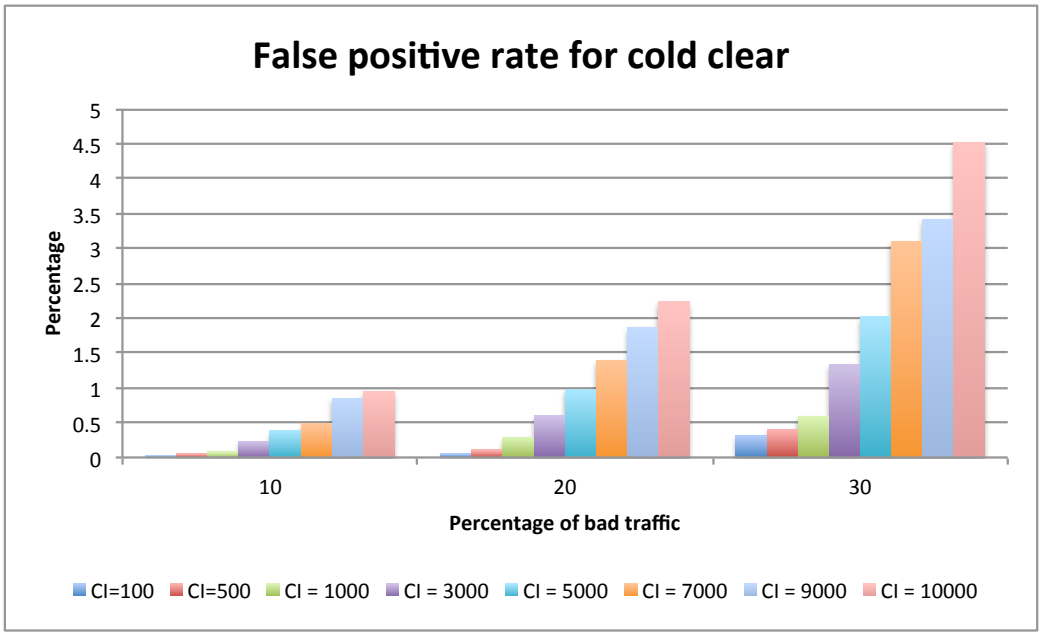
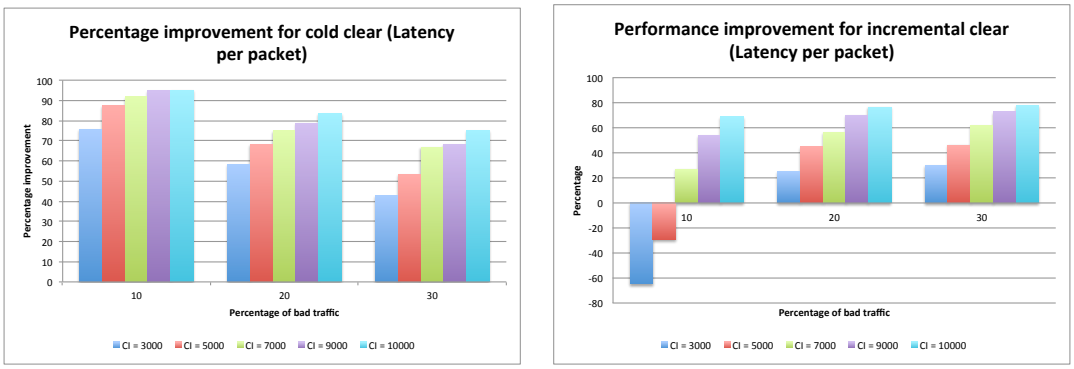


Fig. VI.8. False positive rates with different percentages of bad traffic

VI-C.1. Latency

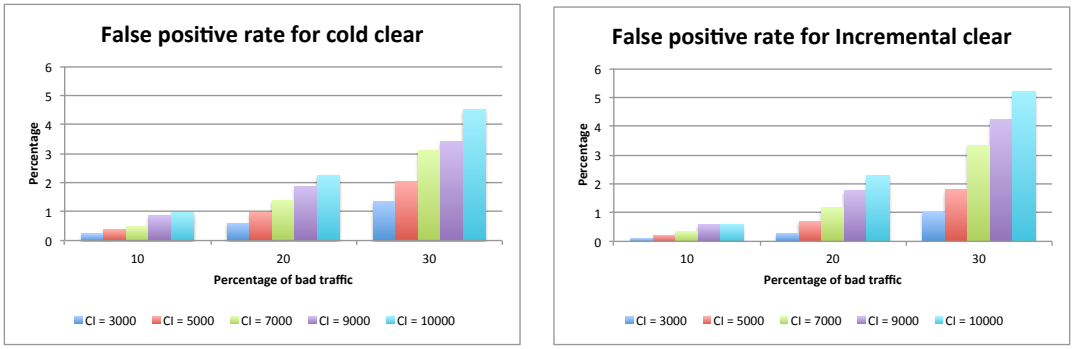


(a) Cold clear bloom filter (b) Incremental clear bloom filter

Fig. VI.9. Comparison of latency

The latencies per packet in cycles have been shown in Figure VI.9. The cold clear setup has a lower latency for most of the cases. But for 30% bad traffic at higher clearing intervals, the incremental clear performs better. This is because the cold clear is completely emptied after the interval and it takes time to re-learn and get back to its previous state before it can be useful. The incremental clear has a static level of occupancy, and therefore the bloom filter is always in a working condition. This allows packets to go through always and therefore latency is slightly lower than in the case of cold clear.

VI-C.2. False Positive



(a) Cold clear bloom filter (b) Incremental clear bloom filter

Fig. VI.10. Comparison of false positive rates

The false positive rates have been compared in the Figure VI.10. The false positive rates are lower in the case of incremental clear for most of the cases. This is because the cold clear clears once in a while and fills up till the limit again before clearing it. But in case of incremental clear, the bits are cleared for every update of the bloom filter. But for higher values of cutoff, it has worse false positive rates than the cold clear. That is because it maintains a very high occupancy rate for the entire time

and therefore the chances of false positive rate is high for the entire simulation.

From the results, we decide that cold clear with clearing interval of 7000 is an optimal solution for the firewall system architecture. Here we see a percentage improvement in latency ranging from 67% to 92% with a false positive rate ranging from 0.5% to 3.1%.

CHAPTER VII

CONCLUSIONS

Firewalls are an integral part of network security. Traditional firewalls must sequentially search through the ACL table, leading to increasing latencies as the number of entries in the table increase. These high latencies ultimately make firewalls vulnerable to denial of service attacks because floods of bad packets can lead to good packets being dropped because of an inability to keep up with the incoming packet bandwidth. This is particularly true for software firewalls implemented in commodity server hardware. In this thesis, we propose a software firewall architecture which removes the sequential ACL lookup from the critical path and thus decreases the latency per packet in the common case. To accomplish this we implement a Bloom filter-based, stochastic pre-classification stage, enabling the bifurcation of the predicted good and predicted bad packet code paths, greatly improving performance. We simulate two different aging mechanisms for the bloomfilter, as well as various intervals for clearing. We run simulations for various percentages of bad traffic. We show that for the optimal case with a cold clear bloomfilter at the interval of 7000 elements, our proposed architecture improves firewall performance 67% to 92% under anonymized trace-based workloads from CAIDA servers. While our approach has the possibility of incorrectly classifying a small subset of bad packets as good, we show that these holes are neither predictable nor permanent, leading to a small, 0.5% to 3.1% of first-time seen, bad packets penetrating the firewall (subsequent packets from the same flows would be filtered out by the access cache).

REFERENCES

[1] Y. Namestnikov, "Kaspersky security bulletin, statistics 2011," http://www.securelist.com/en/analysis/204792216/Kaspersky_Security_Bulletin_Statistics_2011, March 2012.

[2] L. Constantin, "Denial-of-service attacks are on the rise, anti-ddos vendors report," http://www.computerworld.com/s/INPROCEEDINGS/9224028/Denial_of_service_attacks_are_on_the_rise_anti_DDoS_vendors_report, February 2012.

[3] M.G. Gouda and X.-Y.A. Liu, "Firewall design: consistency, completeness, and compactness," in *Proceedings of the 24th International Conference on Distributed Computing Systems*, 2004, pp. 320 – 327.

[4] F. Baboescu and G. Varghese, "Fast and scalable conflict detection for packet classifiers," in *Proceedings of the 10th IEEE International Conference on Network Protocols*, Nov 2002, pp. 270 – 279.

[5] A. Hari, S. Suri, and G. Parulkar, "Detecting and resolving packet filter conflicts," in *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies*, Mar 2000, vol. 3, pp. 1203 –1212.

[6] David Eppstein and S. Muthukrishnan, "Internet packet filter management and rectangle geometry," in *Proceedings of the 12th Annual ACM Symposium on Discrete Algorithms*, Philadelphia, PA, USA, 2001, pp. 827–835.

[7] A.X. Liu and M.G. Gouda, "Firewall policy queries," in *IEEE Transactions on Parallel and Distributed Systems*, June 2009, vol. 20, pp. 766 –777.

[8] Arnaud Launay, "High level firewall language," http://www.hlfl.org, Oct 2003.

[9] Andrew Begel, Steven McCanne, and Susan L. Graham, "Bpf+: exploiting global data-flow optimization in a generalized packet filter architecture," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1999, pp. 123–134.

[10] Yair Bartal, Alain Mayer, Kobbi Nissim, and Avishai Wool, "Firmato: A novel firewall management toolkit," in *ACM Transactions on Computer System*, New York, USA, Nov. 2004, vol. 22, pp. 381–420.

[11] J.D. Guttman, "Filtering postures: local enforcement for global policies," in *Proceedings of IEEE Symposium on Security and Privacy*, May 1997, pp. 120–129.

[12] P. Gupta and N. McKeown, "Algorithms for packet classification," in *IEEE Network*, Mar/Apr 2001, vol. 15, pp. 24 –32.

[13] Paul Francis Tsuchiya and Paul F. Tsuchiya, "A search algorithm for table entries with non-contiguous wildcarding," Unpublished report, Bellcore, Sep 1991.

[14] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," in *SIGCOMM Computer Communication Review*, New York, USA, Oct. 1998, vol. 28, pp. 191–202.

[15] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *SIGCOMM Computer Communication Review*, New York, USA, Aug. 1999, vol. 29, pp. 135–146.

[16] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," in *Micro, IEEE*, Jan/Feb 2000, vol. 20, pp. 34 –41.

[17] F. Chang, Wu chang Feng, and Kang Li, "Approximate caches for packet classification," in *Proceedings of 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, March 2004, vol. 4, pp. 2196 – 2207.

[18] I.L. Chvets and M.H. MacGregor, "Multi-zone caches for accelerating ip routing table lookups," in *Workshop on High Performance Switching and Routing, Merging Optical and IP Technologies*, 2002, pp. 121 – 126.

[19] Kang Li, F. Chang, D. Berger, and Wu chang Feng, "Architectures for packet classification caching," in *Proceedings of the 11th IEEE International Conference on Networks (ICON)*, Sept/Oct 2003, pp. 111 – 117.

[20] Los Gatos CA Robert K. Montoye, "Apparatus for storing 'don't care' in a content addressable memory cell," Patent, US 5319590, HaL Computer Systems Inc., Campbell, CA, USA, 06 1994.

[21] Robert Alan Kempke and Anthony J. McAuley, "Ternary cam memory architecture and methodology," Patent, US 5841874, Motorola Inc., Schaumburg, IL, USA, 11 1998.

[22] Nepean (CA) Garnet Fredrick Randall Gibson, Ottawa (CA) Farhard Shafai, and Kanata (CA) Jason Edward Podaima, "Content addressable memory storage device," Patent, US 6044005, 03 2000.

[23] David E. Taylor, "Survey and taxonomy of packet classification techniques," in *ACM Computing Surveys*, New York, USA, Sep 2005, vol. 37, pp. 238–275.

[24] Burton H. Bloom, "Space/time trade-offs in hash coding with allowable errors," in *Communications of the ACM*, New York, USA, July 1970, vol. 13, pp. 422–426.

[25] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," in *IEEE/ACM Transactions on Networking*, Piscataway, NJ, USA, June 2000, vol. 8, pp. 281–293.

[26] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz, "An architecture for a secure service discovery service," in *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, New York, USA, 1999, pp. 24–35.

[27] S.C. Rhea and J. Kubiatowicz, "Probabilistic location and routing," in *Proceediongs of 21st Annual Joint Conference of the IEEE Computer and Communications Societies*, 2002, vol. 3, pp. 1248 – 1257.

[28] A. Whitaker and D. Wetherall, "Forwarding without loops in icarus," in *Proceedings of IEEE Open Architectures and Network Programming*, 2002, pp. 63 – 75.

[29] Björn Grönvall, "Scalable multicast forwarding," in *SIGCOMM Computer Communication Review*, New York, USA, Jan. 2002, vol. 32, pp. 68–68.

[30] Andrei Broder and Michael Mitzenmacher, "Network applications of bloom filters: A survey," in *Internet Mathematics*, 2002, vol. 1, pp. 636–646.

[31] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood, "The gem5 simulator," in *SIGARCH Computer Architecture News*, New York, USA, Aug. 2011, vol. 39, pp. 1–7.

[32] KC. Claff, Dan Andersen, and Paul Hick, "The caida anonymized 2012 internet traces," http://www.caida.org/data/passive/passive_2012_dataset.xml, Feb 2012.